Optimization

Device management, Optimization

GPU Programming

http://cuda.nik.uni-obuda.hu

Szénási Sándor
szenasi.sandor@nik.uni-obuda.hu

GPU Education Center of Óbuda University

Device Management

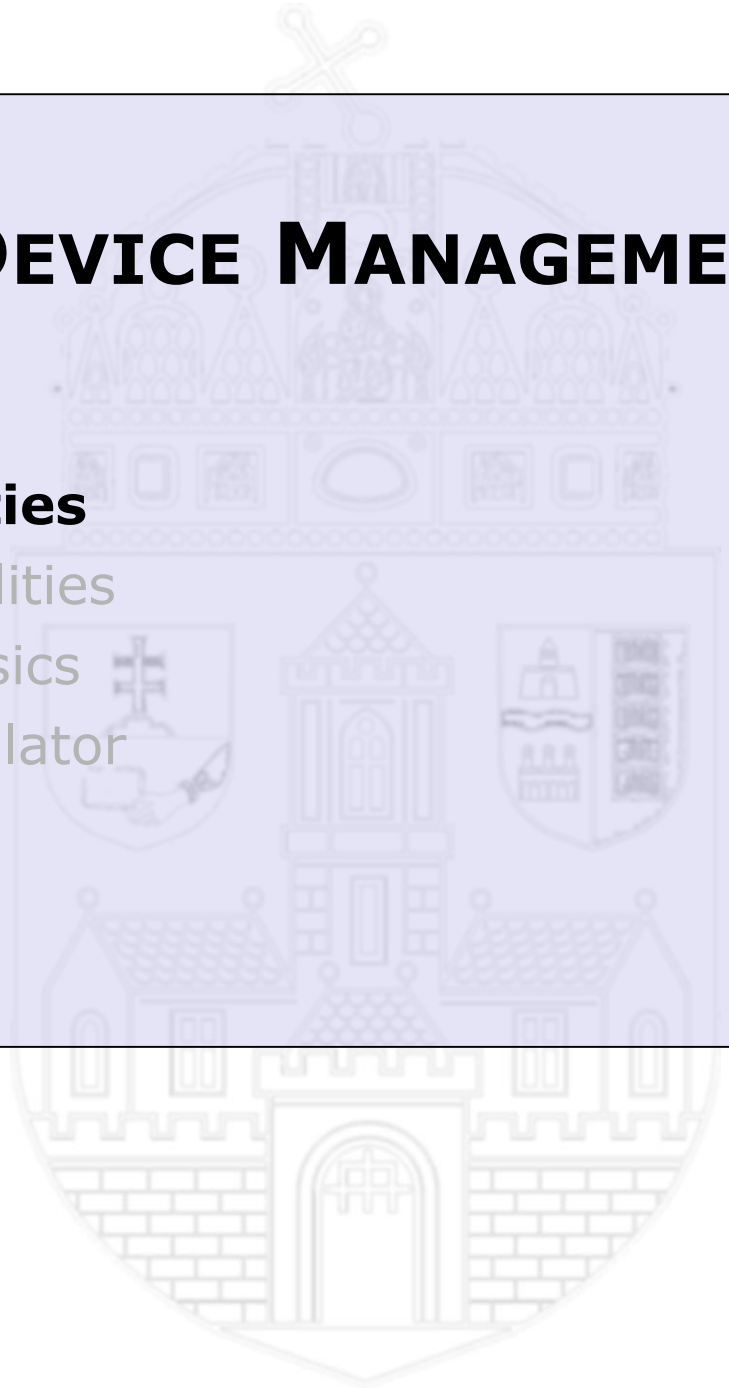
Device properties

Compute capabilities

Optimization basics

Occupancy calculator

Available tools



Device management

Number of CUDA compatible devices

- The result of the *cudaGetDeviceCount* function is the number of CUDA-available devices

```
1  int deviceCount;
2  cudaGetDeviceCount(&deviceCount);
```

- The function will store the number of CUDA compatbile devices into the passed deviceCount variable

Select the active CUDA compatible device

- This function is used to select the device associated to the host thread. A device must be selected before any __global__ function or any function from the runtime API is called
- The parameter of this function is the number of the selected device (numbering starts with 0)

```
1  int deviceNumber = 0;
2  cudaSetDevice(deviceNumber);
```

- Missing the function call, the framework will automatically select the first available CUDA device
- The result of the function will affect the entire host thread

Detailed information about devices

The CUDA framework contains a class structure named *cudaDeviceProp*, to store the detailed information of the devices. The main fields of this structure are:

cudaDeviceProp structure	
name	Name of the device
totalGlobalMem	Size of the global memory
sharedMemPerBlock	Size of the shared memory per block
regsPerBlock	Number of registers per block
totalConstMem	Size of the constant memory
warpSize	Size of the warps
maxThreadsPerBlock	Maximum number of threads by block
maxThreadsDim	Maximum dimension of thread blocks
maxGridSize	Maximum grid size
clockRate	Clock frequency
minor, major	Version numbers
multiprocessorCount	Number of multiprocessors
deviceOverlap	Is the device capable to overlapped read/write

Acquire the detailed information about devices

- The result of the *cudaGetDeviceProperties* is the previously introduced cudaDeviceProp structure.
- The first parameter of the function is a pointer to an empty cudaDevireProp structure. The second parameter is the identifier of the device (numbering starts with 0)

```
1   int deviceNumber = 1;
2   cudaDeviceProperty deviceProp;
3   cudaGetDeviceProperties(&deviceProp, deviceNumber);
```

Exam

Write out the number of available devices.
List the number of these devices.
List the detailed data of an user selected device.

Device Management

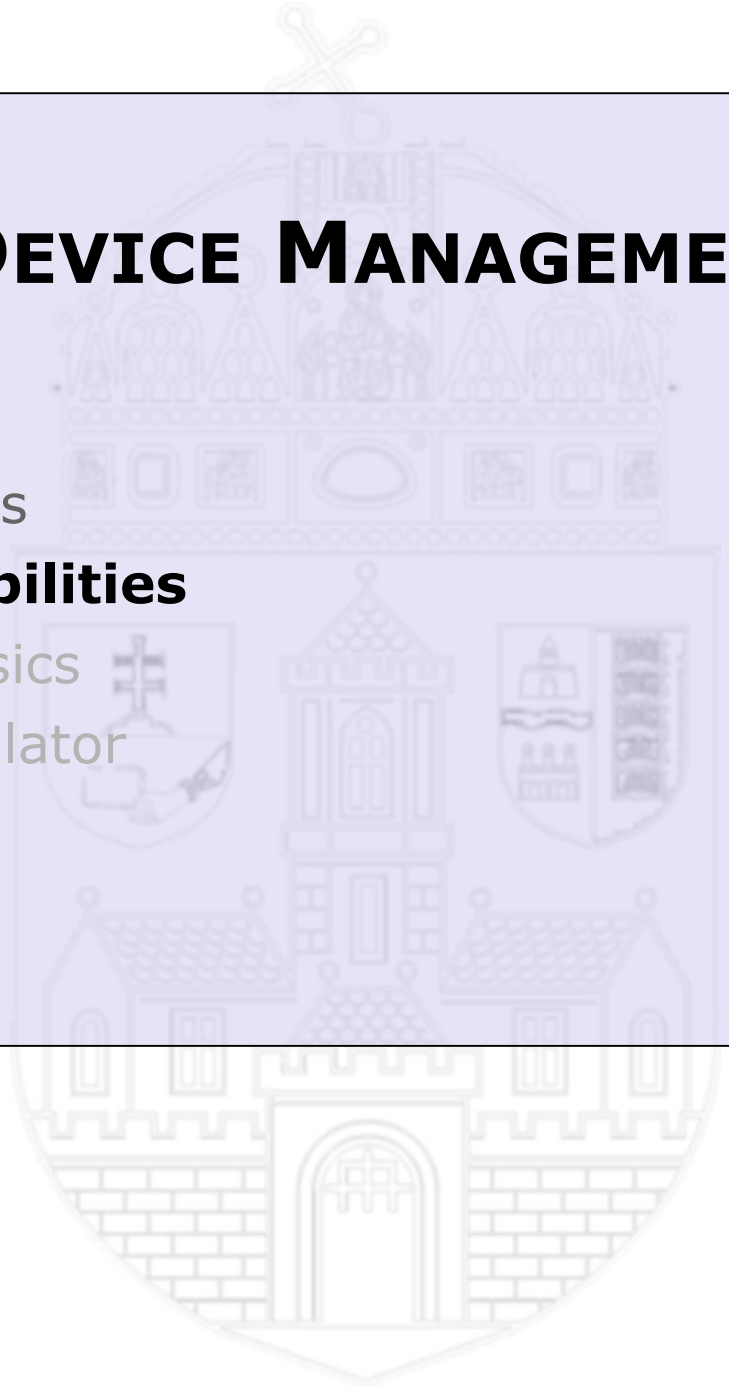
Device properties

Compute capabilities

Optimization basics

Occupancy calculator

Available tools



Compute capability (1)

- The difference between the newer and older graphics cards are more than the number of execution units and the speed of the processing elements. Often there are really dramatic changes in the whole CUDA architecture. The compute capability is a sort of hardware version number.
- The compute capability of a device is defined by a major revision number and a minor revision number.
- Devices with the same major revision number are of the same core architecture

Details for hardware versions

- Compute capability 1.0
  - The maximum number of threads per block is 512
  - The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512,and 64, respectively
  - The maximum size of each dimension of a grid of thread blocks is 65535
  - The warp size is 32 threads
  - The number of registers per multiprocessor is 8192
  - The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks
  - The total amount of constant memory is 64 KB
  - The cache working set for constant memory is 8 KB per multiprocessor

Compute capability (2)

- Compute capability 1.0 (cont.)
  - The cache working set for constant memory is 8 KB per multiprocessor
  - The cache working set for texture memory varies between 6 and 8 KB per multiprocessor
  - The maximum number of active blocks per multiprocessor is 8
  - The maximum number of active warps per multiprocessor is 24
  - The maximum number of active threads per multiprocessor is 768
  - For a texture reference bound to a one-dimensional CUDA array, the maximum width is 213
  - For a texture reference bound to a two-dimensional CUDA array, the maximum width is 216 and the maximum height is 215
  - For a texture reference bound to linear memory, the maximum width is 227
  - The limit on kernel size is 2 million PTX instructions
  - Each multiprocessor is composed of eight processors, so that a multiprocessor is able to process the 32 threads of a warp in four clock cycles
- Compute capability 1.1
  - Support for atomic functions operating on 32-bit words in global memory

Compute capability (3)

- Compute capability 1.2
  - Support for atomic functions operating in shared memory and atomic functions operating on 64-bit words in global memory
  - Support for warp vote functions
  - The number of registers per multiprocessor is 16384
  - The maximum number of active warps per multiprocessor is 32
  - The maximum number of active threads per multiprocessor is 1024
- Compute capability 1.3
  - Support for double-precision floating-point numbers
- Compute capability 2.0
  - 3D grid of thread blocks
  - Floating point atomic functions (addition)
  - __ballot() function is available (warp vote)
  - __threadfence_system() function is available
  - __systhreads_count() function is available
  - __systhreads_and() function is available
  - __systhreads_or() function is available
  - Maximum dimension of a block is 1024
  - Maximum number of threads per block

Compute capability (4)

- Compute capability 2.0 (cont)
  - Warp size is 32
  - Maximum threads per multiprocessors is 1536
  - Number of 32 bit registers per multiprocessors is 32K
  - Number of shared memory banks is 32
  - Amount of local memory per thread is 512KB
- Compute capability 3.0
  - Atomic functions operating on 64-bit integer values in shared memory
  - Atomic addition operating on 32-bit floating point values in global and shared memory
  - __ballot()
  - __threadfence_system()
  - __syncthreads_count()
  - __syncthreads_and()
  - __syncthreads_or()
  - Surface functions
  - 3D grid of thread blocks
  - Maximum number of resident blocks per multiprocessor is 16
  - Maximum number of resident warps per multiprocessor is 64
  - Maximum number of resident threads per multiprocessor is 2048

Compute capability (5)

- Compute capability 3.0 (cont)
  - Number of 32-bit registers per multiprocessor is 64K
- Compute capability 3.5
  - Funnel Shift
  - Maximum number of 32-bit registers per thread is 255

Device parameters (1)

Device name	Number of MPs	Compute capability
GeForce GTX 280	30	1.3
GeForce GTX 260	24	1.3
GeForce 9800 GX2	2x16	1.1
GeForce 9800 GTX	16	1.1
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 8800 GT	14	1.1
GeForce 9600 GSO, 8800 GS, 8800M GTX	12	1.1
GeForce 8800 GTS	12	1.0
GeForce 8500 GT, 8400 GS, 8400M GT, 8400M GS	2	1.1
GeForce 8400M G	1	1.1
Tesla S1070	4x30	1.3
Tesla C1060	30	1.3
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 1000 Model S4	4x16	1.0
Quadro FX 1700, FX 570, NVS 320M, FX 1600M	4	1.1
GeForce GTX 480	15	2.0
GeForce GTX 470	14	2.0

Device parameters (2)

Device name	Compute capability
GeForce GT 610	2.1
GeForce GTX 460	2.1
GeForce GTX 560 Ti	2.1
GeForce GTX 690	3.0
GeForce GTX 670MX	3.0
GeForce GT 640M	3.0
Tesla K20X, K20	3.5

- More details can be found at

http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units

Device Management

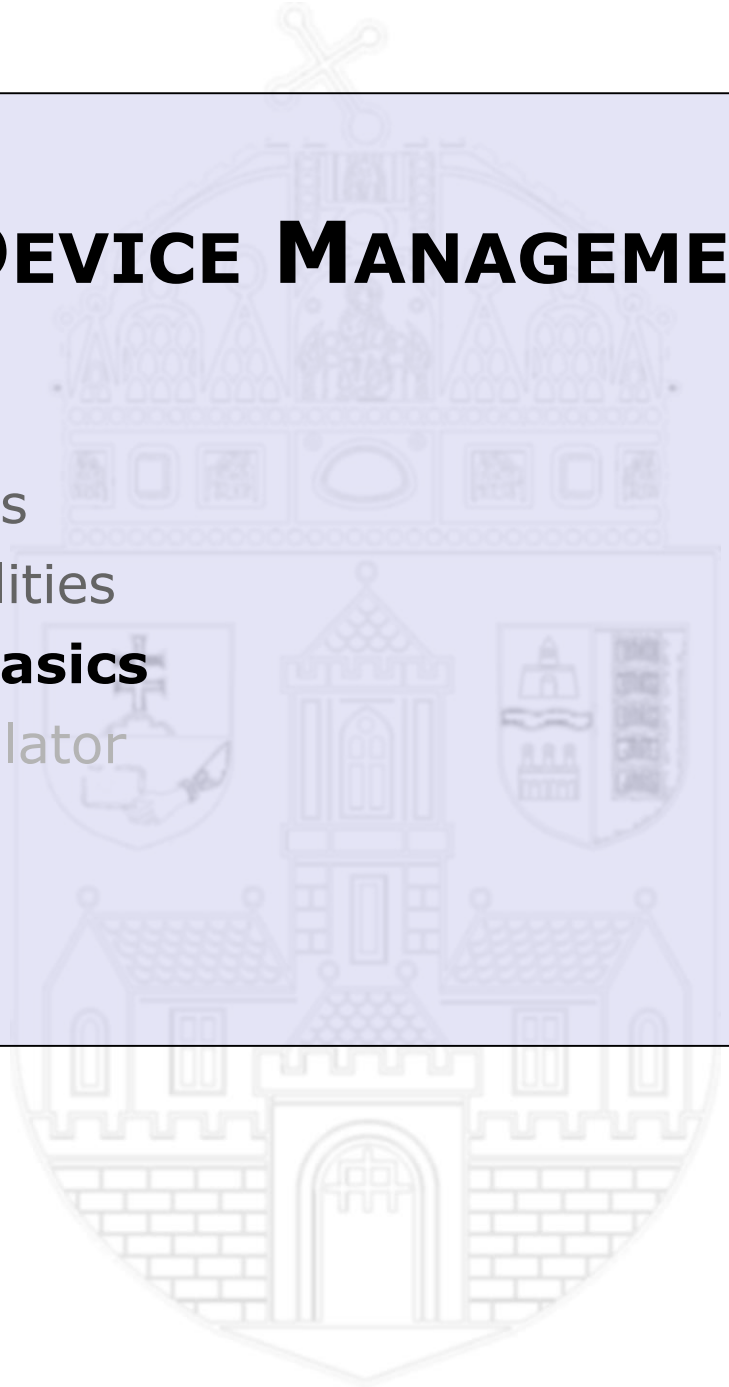
Device properties

Compute capabilities

Optimization basics

Occupancy calculator

Available tools



Device Management

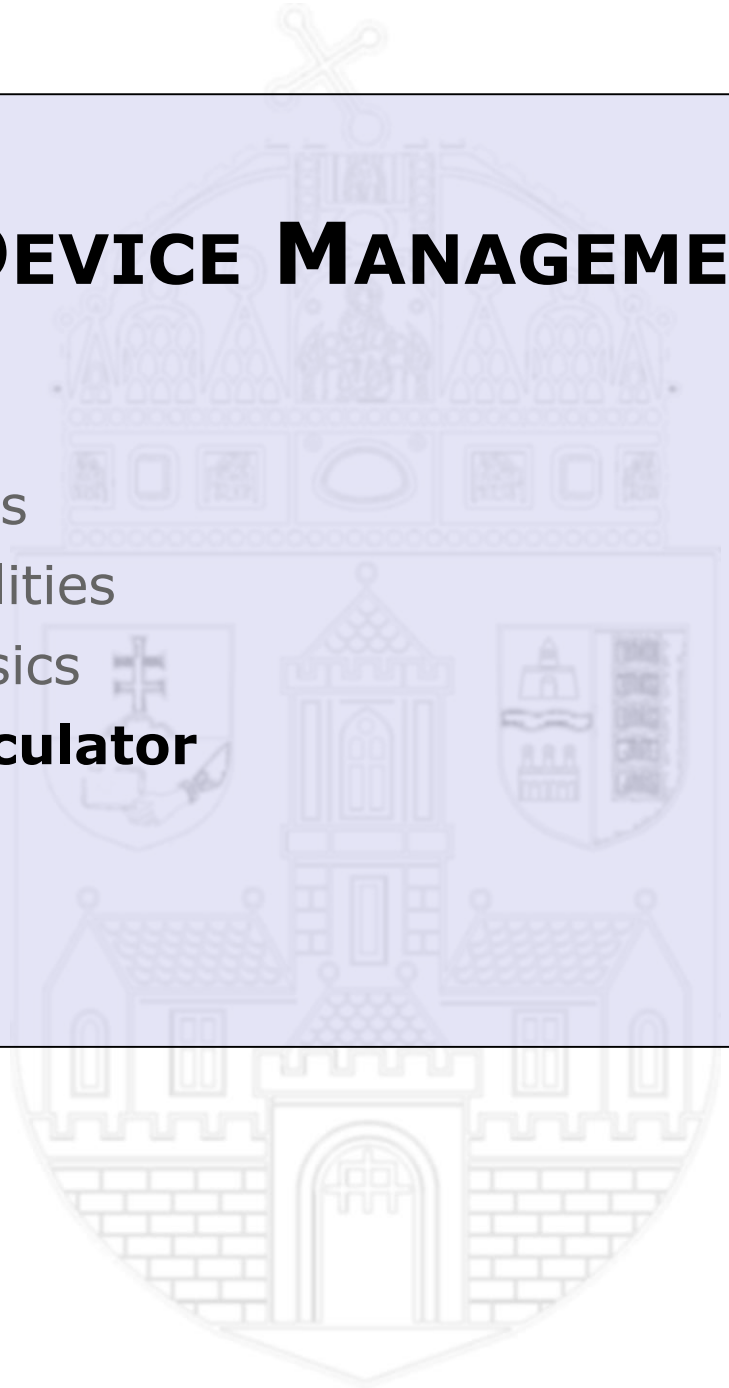
Device properties

Compute capabilities

Optimization basics

Occupancy calculator

Available tools



Execution overview

- Problem space is divided into blocks
  - Grid is composed of independent blocks
  - Blocks are composed of threads
- Instructions are executed per warp
  - In case of Fermi, 32 threads form a warp
  - Fermi can have 48 active warps per SM (1536 threads)
  - Warp will stall if any of the operands is not ready
- To avoid latency
  - Switch between contextes while warps stalled
  - Context switching latency is very small
- Registers and shared memory are allocated for a block as long as the block is active
  - Once a block is active it will stay active until all threads completed in that block
  - Registers/shared memory do not need store/reload in case of context switching

Occupancy

- Occupancy is the ratio of active processing units to available processing units
  Occupancy = Active Warps / Maximum Number of Warps
- Occupancy is limited by:
  - Max Warps or Max Blocks per Multiprocessor
  - Registers per Multiprocessor
  - Shared memory per Multiprocessor

- Occupancy = Min(register occ., shared mem occ., block size occ.)

Occupancy and registers

- Fermi has 32K registers per SM
- The maximum number of threads is 1536
- For example, if a kernel uses 40 registers per thread:
  - Number of active threads: 32K / 40 = 819
  - Occupancy: 819 / 1536 = 0,53
- In this case the number of registers limits the occupancy (meanwhile there are some unused resources in the GPU)
- Goal: try to limit the register usage
  - Check register usage: compile with –ptxax-optoins=-v
  - Limit register usage: compile with –maxregcount
- For example, in case of 21 registers:
  - Number of active threads: 32K / 21 = 1560
  - Occupancy: 1560 / 1536 = ~1
  - This means only that the number of registers will not limit the occupancy (it is higly depends on other resources)

Occupancy and shared memory

- Size of shared memory is configurable in Fermi
  - 16K shared memory
  - 48K shared memory (we use this configuration in the examples)
- For example, if a kernel uses 64 bytes of shared memory
  - Number of active threads: 48K / 64 = 819
  - Occupancy: 819 / 1536 = 0,53
- In this case the size of shared memory limits the occupancy (meanwhile there are some unused resources in the GPU)
- Goal: try to limit the shared memory usage
  - Check shared memory usage: compile with –ptxax-optoins=-v
  - Limit shared memory usage
    - Use lower shared memory in kernels (kernel invocation)
    - Use appropriate L1/Shared configuration in case of Fermi
- For example, in case of 32 bytes of shared memory:
  - Number of active threads: 48K / 32 = 1536
  - Occupancy: 1536 / 1536 = 1
  - This means only that the size of shared memory will not limit the occupancy (it is higly depends on other resources)

Occupancy and block size

- Each SM can have up to 8 active blocks
- There is a hardware based upper limit for block size
  - Compute Capability 1.0 – 512
  - Compute Capability 2.0 – 1024
- Lower limit is 1 but small block size will limit the total number of threads
- For example,
  - Block size: 128
  - Active threads in one SM: 128 * 8 = 1024
  - Occupacy: 1536 / 1024 = 0,66
- In this case the block size limits the occupancy (meanwhile there are some unused resources in the GPU)
- Goal: try to increase the block size (kernel invocation parameter)
- For example,
  - Block size: 192
  - Active threads in one SM: 192 * 8 = 1536
  - Occupacy: 1536 / 1536 = 1

CUDA Occupancy calculator

- A CUDA tool to investigate the occupancy
- In practice it is an Excel sheet, located in *„NVIDIA GPU Computing SDK x.x\C\tools\CUDA_Occupancy_Calculator.xls”*
- Input data:
  - Hardware configuration
    - Compute Capability
    - Shared Memory Config
  - Resource usage
    - Threads per block
    - Registers per thread
    - Shared memory per block
- Output data:
  - Active threads per MP
  - Active warps per MP
  - Active thread blocks per MP
  - Occupancy of each MP

CUDA Occupancy calculator - example

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	2,0	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	16	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1536	(Help)
Active Warps per Multiprocessor	48	
Active Thread Blocks per Multiprocessor	6	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	2,0
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	64
Register allocation granularity	warp
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity (for block register allocation)	0
Maximum Thread Block Size	1024

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (≈Registers Per Thread * Threads Per Block)	4096	32768	8
Shared Memory (Bytes)	4096	49152	12

Note: SM is an abbreviation for (Streaming) Multiprocessor

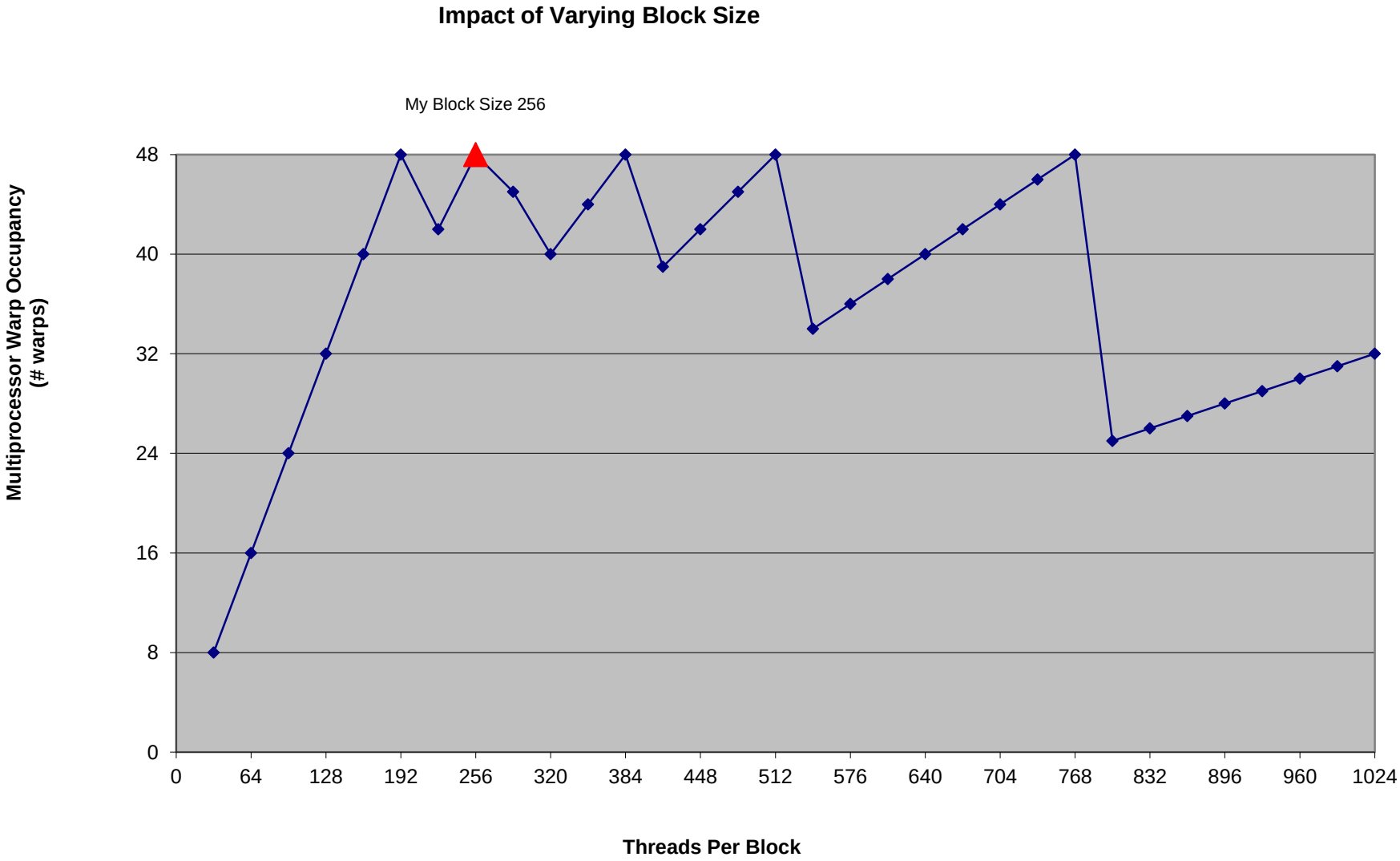
Hardware configuration

Resource usage

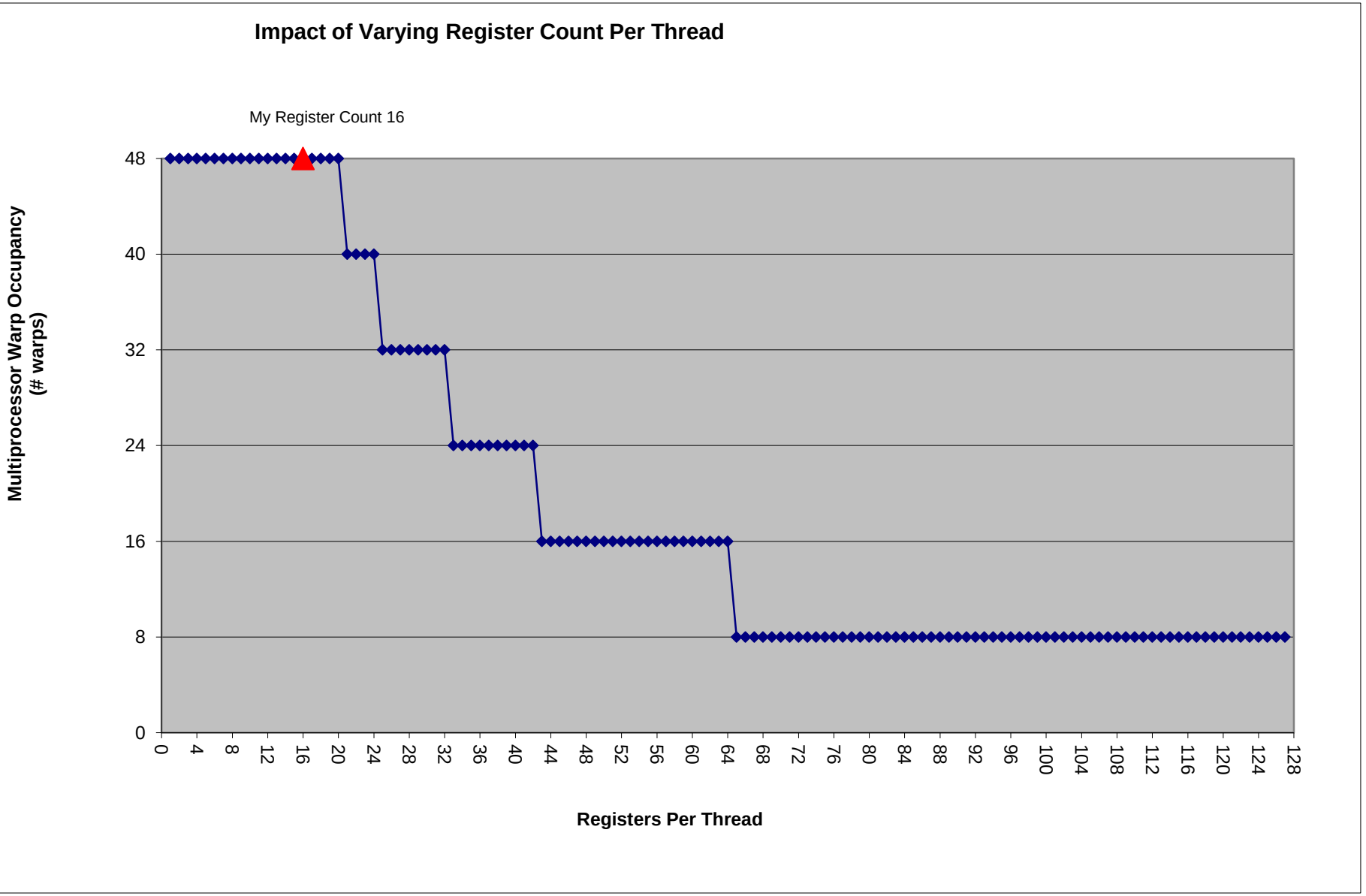
Occupancy details

Physical limits

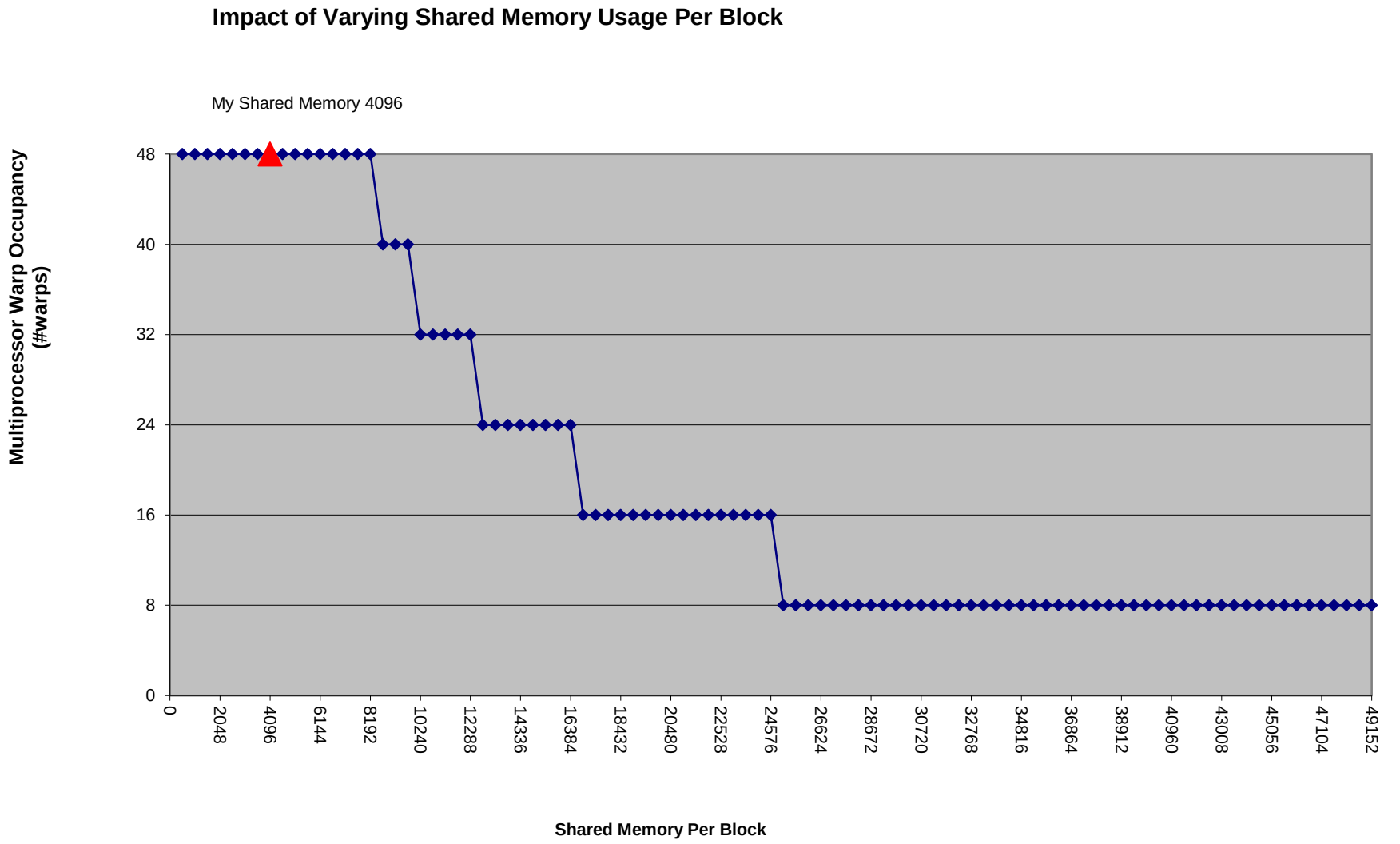
CUDA Occupancy calculator – impact of varying block size



CUDA Occupancy calculator – impact of varying register count



CUDA Occupancy calculator – impact of varying shared memory



Block size considerations [18]

- Choose number of threads per block as a multiple of warp size
- Avoid wasting computation on under-populated warps
- Optimize block size
  - More thread block – better memory latency hiding
  - Too much thread block – fewer register per thread, kernel invocation can fail if too many are registers are used
- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough registers to compile and invoke successfully
  - This all depends on your computation!
    - Experiment!
- Try to maximize occupancy
  - Increasing occupancy does not necessarily increase performance
  - But, low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels

Device Management

Device properties

Compute capabilities

Optimization basics

Occupancy calculator

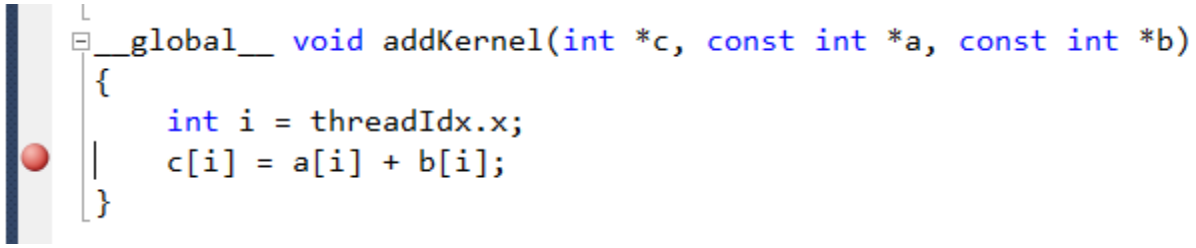
Available tools

Parallel Nsight

- Debugger for GPGPU development
- Available only for registered users(?):
  http://www.nvidia.com/object/nsight.html
- Available editions
  - Visual Studio Edition
    https://developer.nvidia.com/nvidia-nsight-visual-studio-edition
  - Nsight Eclipse Edition
- Main features
  - Visual Studio/Eclipse support
  - PTX/SASS Assembly Debugging
  - CUDA Debugger (debug kernels directly)
  - Use conditional breakpoints
  - View GPU memory
  - Graphics debugger
  - Profiler functions
- Hardware requirements
  - Analyzer -Single GPU system
  - CUDA Debugger –Dual GPU system
  - Direct3D Shader Debugger –Two separate GPU systems

Kernel debugging

- Main steps for local debugging
  - Start Nsight Monitor
    (All Programs > NVIDIA Corporation > Nsight Visual Studio Edition 2.2 > Nsight Monitor)
  - Set breakpoint
    Like setting breakpoint in CPU code



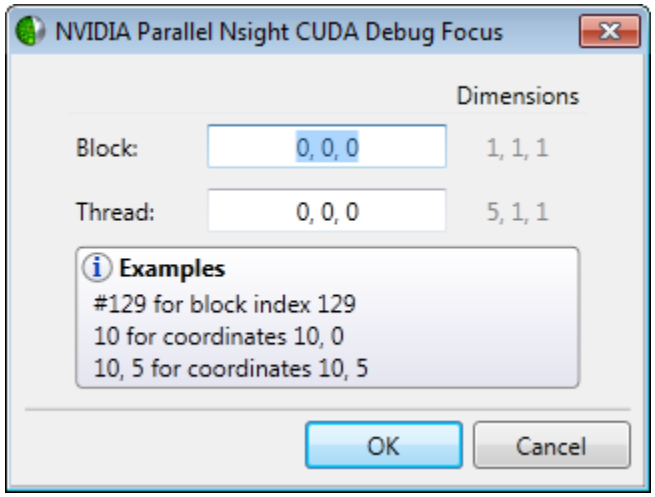
  - Start CUDA debugging in Visual Studio
    (Nsight/Start CUDA debugging)
  - Debugger will stop at the breakpoint
  - All the common debugger commands are available
    - Step over
    - Step into
    - Etc.
- Remote debugging
  - We do not discuss

Watch GPU memory regions

- Nsight supports the Visual Studio „Memory” window for examining the contents of GPU memory
  - Shared memory
  - Local memory
  - Global memory
- To show a memory region, select *Debug/Windows/Memory*
  - In case of kernel debugging just enter the name of the variable of the direct address
  - In case of direct addresses use the following keywords: __shared__, __local__, __device__
  - For example: (__shared__ float*)0
- The common visual studio functions also available
  - Watch window to check kernel variables
  - Move the cursor over a variable to see the actual value
- Built-in CUDA variables are also available
  - threadIdx
  - blockIdx
  - blockDim
  - gridDim
  - etc.

CUDA Debug Focus

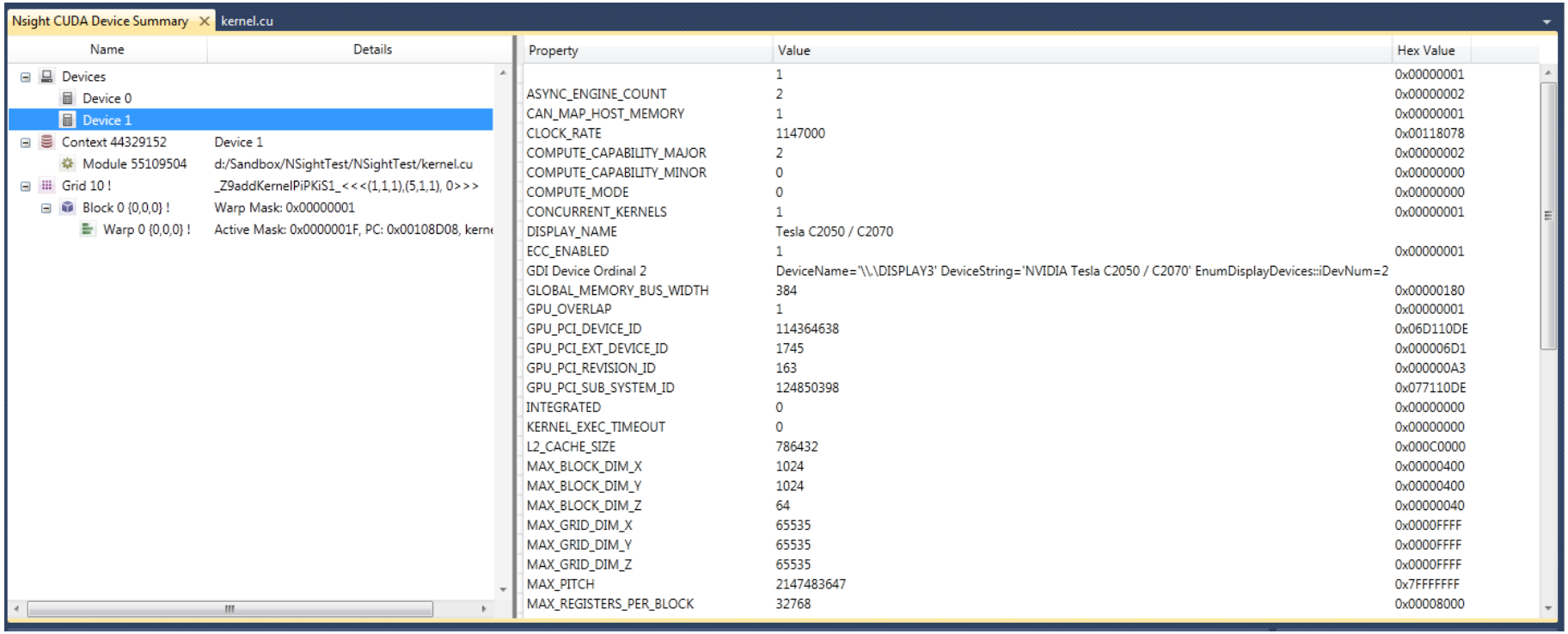
- Some variables in CUDA belongs to a context
  - Registers and local memory to threads
  - Shared memory to blocks
- To see the variable actual value the developer must define the owner thread (block index and thread index)
  - Select *Nsight/Windows/CUDA Debug Focus*



  - Set block index
  - Set thread index
- Watch window/quick watch etc. will show information about the variables of the corresponding thread

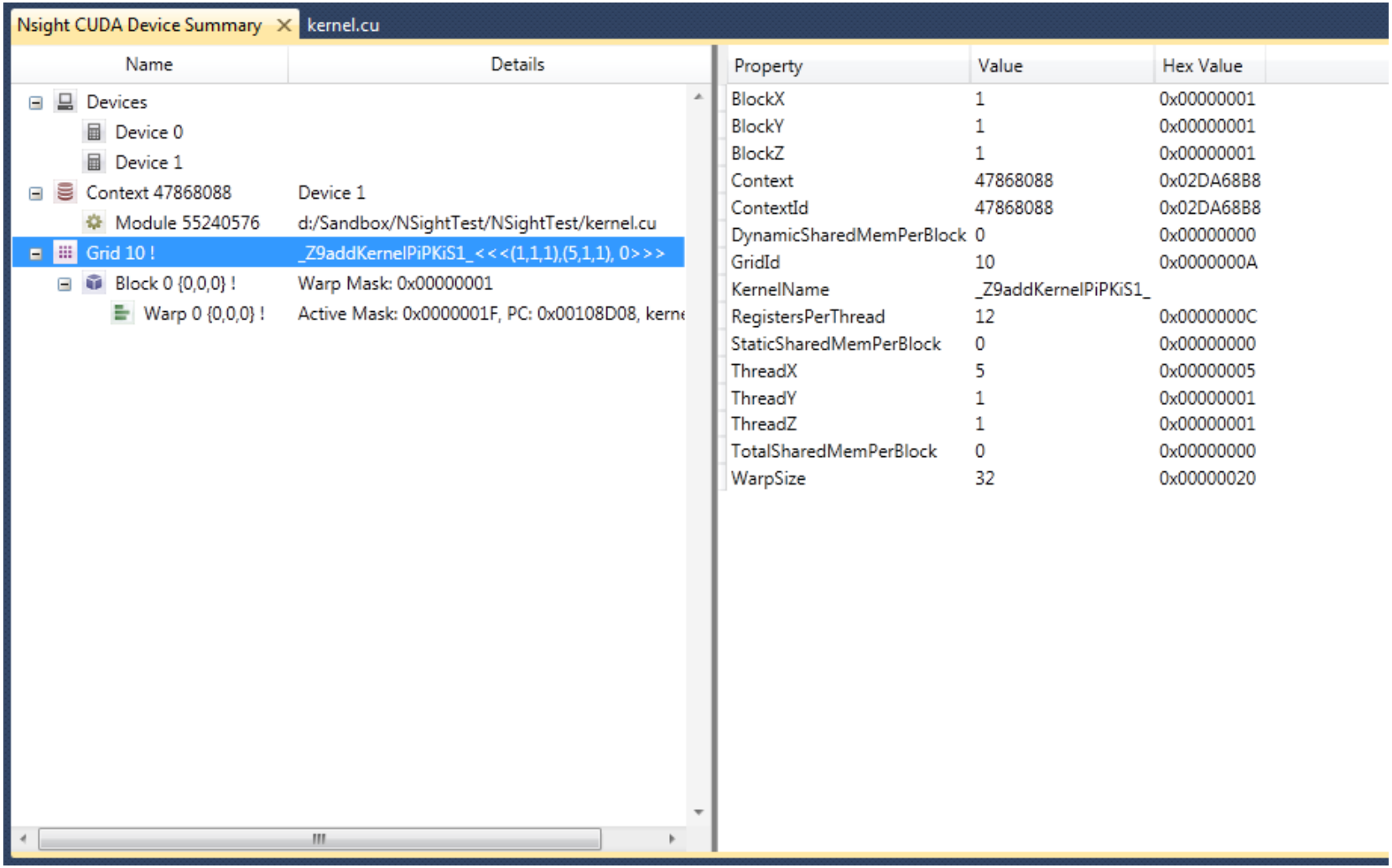
CUDA Device Summary

- An overview about the state of available devices
  - Select *Nsight/Windows/CUDA Device Summary*
  - Select a device from the list
  - Lots of statis and runtime parameters are displayed in the right



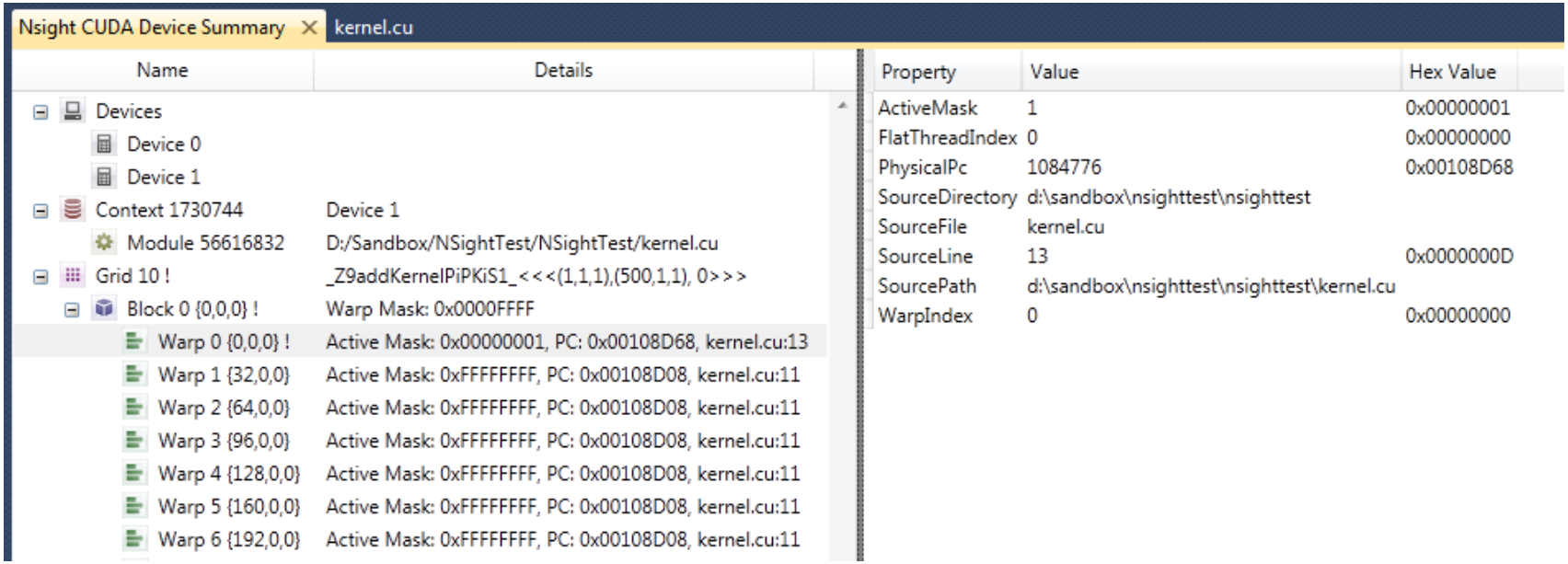
CUDA Device Summary - grid

- An overview about the state of available devices
  - Select *Nsight/Windows/CUDA Device Summary*
  - Select a grid from the list



CUDA Device Summary - warp

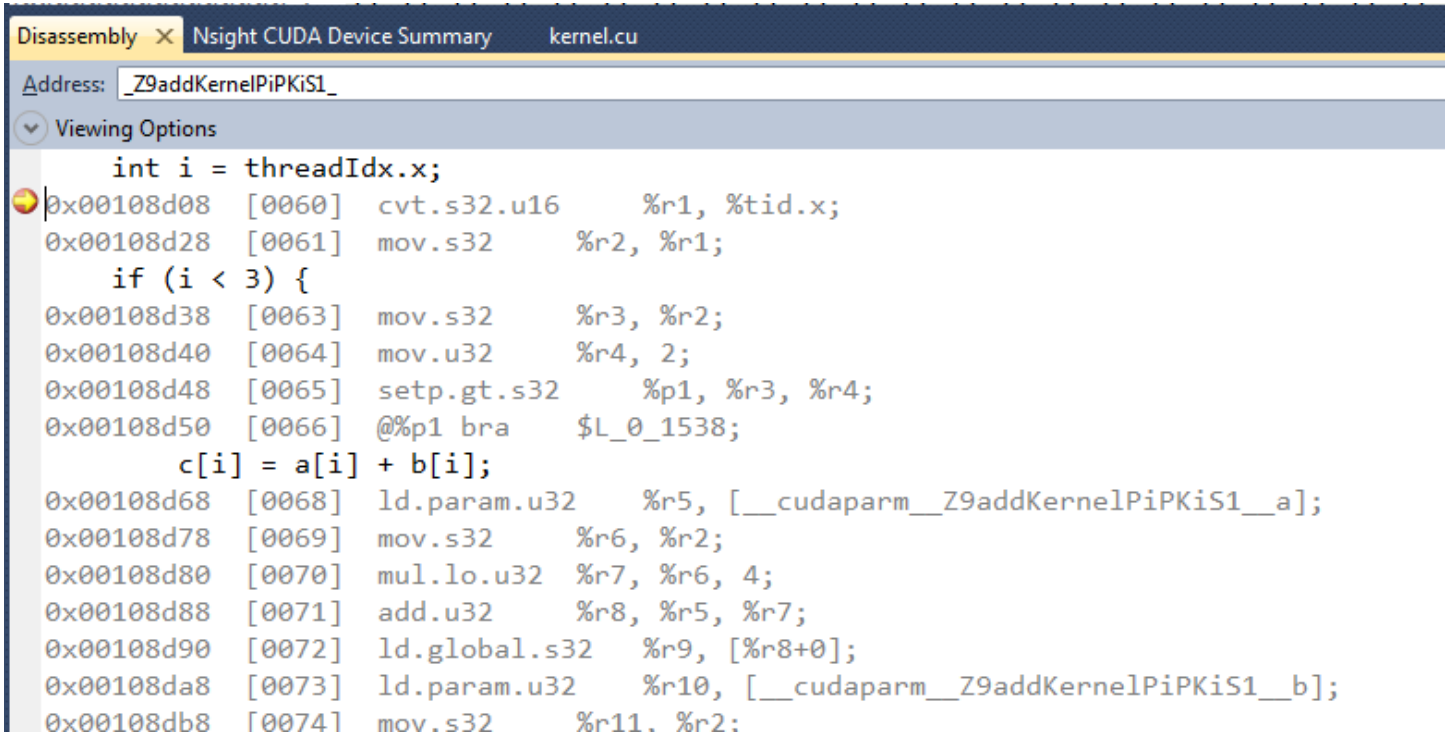
- An overview about the state of available devices
  - Select *Nsight/Windows/CUDA Device Summary*
  - Select a running warp



- Developer can check the current state of all running warps
- *SourceFile/SourceLine* can be very usefull to understand the execution mechanism

Debugging PTX code

- Check the Tools/Options/Debugging options
  - Select "Enable Address Level Debugging"
  - Select "Show disassembly if source is not available"
- When the CUDA debugger is stopped
  - Select "Go to Disassembly"
  - The PTX code appears (SASS code is also available)
- Debugging is the same as CPU applications



Using the memory checker

- The CUDA Memory Checker detects problems in global and shared memory. If the CUDA Debugger detects an MMU fault when running a kernel, it will not be able to specify the exact location of the fault. In this case, enable the CUDA Memory Checker and restart debugging, and the CUDA Memory Checker will pinpoint the exact statements that are triggering the fault [22]
- Select Nsight/Options/CUDA
  - Set "Enable Memory Checker" to true
- Launch the CUDA debugger and run the application
  - During the execution if the kernel tries to write to an invalid memory location (for example in case of arrays) the debugger will stop
  - The debugger will stop *before* the execution of this instruction
- The CUDA memory checker will write results to the Output window
  - Launch parameters
  - Number of detected problems
  - GPU state in these cases
    - Block index
    - Thread index
    - Sourcecode line number
  - Summary of access violations

CUDA memory checker result

```
=========================================================================================
CUDA Memory Checker detected 2 threads caused an access violation:
Launch Parameters
   CUcontext     = 003868b8
   CUstream      = 00000000
   CUmodule      = 0347e780
   CUfunction    = 03478980
   FunctionName = _Z9addKernelPiPKiS1_
   gridDim       = {1,1,1}
   blockDim      = {5,1,1}
   sharedSize    = 0
   Parameters:
   Parameters (raw):
       0x05200000 0x05200200 0x05200400
GPU State:
  Address  Size     Type      Block  Thread       blockIdx  threadIdx     PC  Source
-----------------------------------------------------------------------------------
 05200018    4    adr st         0      3       {0,0,0}   {3,0,0}  0000f0  d:\sandbox\nsighttest\nsighttest\kernel.cu:12
 05200020    4    adr st         0      4       {0,0,0}   {4,0,0}  0000f0  d:\sandbox\nsighttest\nsighttest\kernel.cu:12

Summary of access violations:
=========================================================================================

Parallel Nsight Debug
Memory Checker detected 2 access violations.
error = access violation on store
blockIdx = {0,0,0}
threadIdx = {3,0,0}
address = 0x05200018
accessSize = 4
```

Possible error codes and meanings

- CUDA memory checker error codes:

CUDA memory checker error codes	
mis ld	misaligned access during a memory load
mis st	misaligned access during a memory store
mis atom	misaligned access during an atomic memory transaction - an atomic function was passed a misaligned address
adr ld	invalid address during a memory load
adr st	invalid address during a memory store -attempted write to a memory location that was out of range, also sometimes referred to as a limit violation.
adr atom	invalid address during an atomic memory transaction - an atomic function attempted a memory access at an invalid address.

Matrix multiplication

Exam

- Create CUDA application to solve the following problems: multiplying 2 dimensional (NxN) matrices with the GPU
  - N is a constant in source code
  - Allocate memory for 3 NxN matrices(A, B, C)
  - Fill the A matrix with numbers (for example: $a_{i,j} = i + j$)
  - Fill the B matrix with numbers (for example: $b_{i,j} = i - j$)
  - Allocate 3 NxN matrices in the global memory of the graphics card (devA, devB, devC)
  - Move the input date to the GPU: A → devA, B → devB
  - Execute a kernel to calculate devC = devA * devB
  - Move the results back to the system memory: devC → C
  - List the values in the C vector to the screen

Optimization strategies

- Memory usage
  - Use registers
  - Use shared memory
  - Minimize CPU-GPU data transfers
  - Processing data instead of moving it (move code to the GPU)
  - Group data transfers
  - Special memory access patterns (we don’t discuss)
- Maximize parallel execution
  - Maximize GPU parallelism
    - Hide memory latency by running as many threads as possible
  - Use CPU-GPU parallelism
  - Optimize block size
  - Optimize number of blocks
  - Use multiple-GPUs
- Instruction level optimization
  - Use float arithmetic
  - Use low precision
  - Use fast mathematic functions
  - Minimize divergent warps
    - Branch conditions

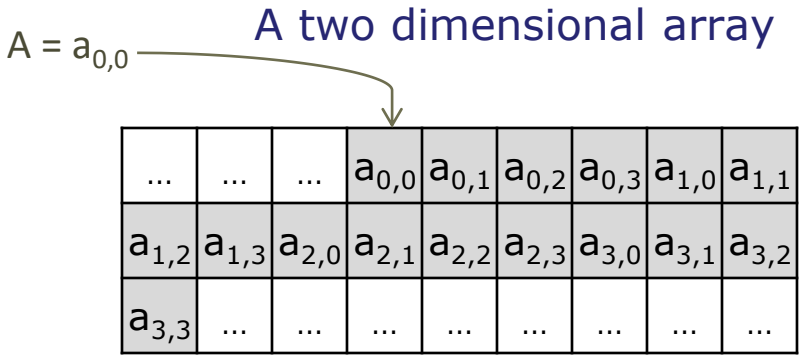
Multi-dimensional matrix in global memory

- We can use multi-dimensional arrays in C programs, but these are obviously stored in a linear memory area
- For example a 4x4 matrix in the memory:

A matrix

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

A two dimensional array



Access elements of a multi-dimensional array

- We know the address of the first item in the array and we know the size of each elements. In this case we can use the following formula:
  $a_{row,col} = a_{0,0} + (row * col\_number + col) * item\_size$
- The CUDA kernel will get only the starting address of the array, we have to use this formula to access the elements

Multi-dimensional matrix multiplication

- If one thread processes one item in the matrix, we need as many threads as the number of matrix elements. A relatively small 30x30 matrix needs 900 threads in GPU, therefore we have to use multiple blocks
- Therefore we have to use the block identifier in the kernel. The improved kernel for the devC = devA * devB matrix multiplication:

```
1  __global__ static void MatrixMul(float *devA, float *devB, float *devC) {
2     int indx = blockIdx.x * blockDim.x + threadIdx.x;
3     int indy = blockIdx.y * blockDim.y + threadIdx.y;
4
5     if (indx < N && indy < N) {
6        float sum = 0;
7        for(int i = 0; i < N; i++) {
8           sum += devA[indy * N + i] * devB[i * N + indx];
9        }
10       devC[indy * N + indx] = sum;
11    }
12 }
```

Multi-dimensional matrix in the GPU memory

- Initialization, memory allocation

```
1  cudaSetDevice(0);
2  float A[N][N], B[N][N], C[N][N]; float *devA, *devB, *devC;
3  cudaMalloc((void**) &devA, sizeof(float) * N * N);
4  cudaMalloc((void**) &devB, sizeof(float) * N * N);
5  cudaMalloc((void**) &devC, sizeof(float) * N * N);
```

- Move input data

```
6  cudaMemcpy(devA, A, sizeof(float) * N * N, cudaMemcpyHostToDevice);
7  cudaMemcpy(devB, B, sizeof(float) * N * N, cudaMemcpyHostToDevice);
```

- Invoke the kernel

```
8  dim3 grid((N - 1) / BlockN + 1, (N - 1) / BlockN + 1);
9  dim3 block(BlockN, BlockN);
10 MatrixMul<<<grid, block>>>(devA, devB, devC);
11 cudaThreadSynchronize();
```

- Move the results back, free memory

```
12 cudaMemcpy(C, devC, sizeof(float) * N * N, cudaMemcpyDeviceToHost);
13 cudaFree(devA); cudaFree(devB); cudaFree(devC);
```

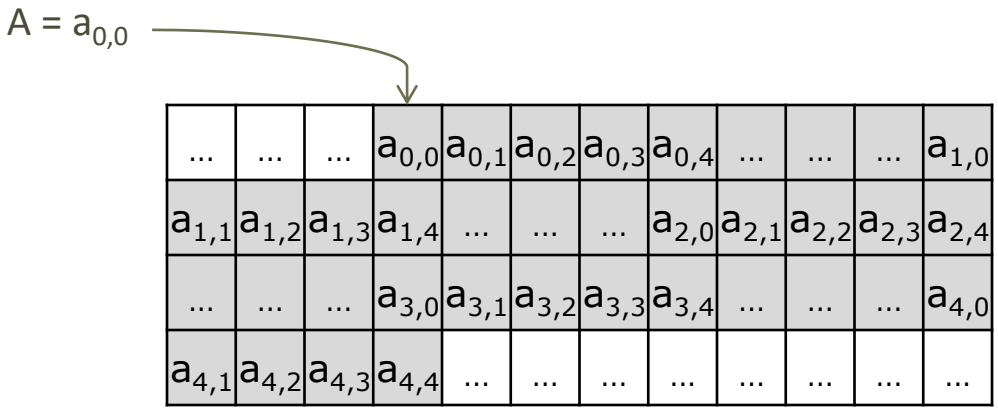
Aligned arrays

- In some cases the number of columns in one matrix row differs from the size of the rows in the memory. This can speed up the access of values because technical reasons (for example with the real memory row size, we can use faster multiplications or we can utilize the capacity of the GPU memory controllers)
- A simple 5x5 matrix with 8 item alignment:

A matrix

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$

A array in memory



Access elements in case of aligned storage

- The formula is similar but we use the aligned row size:
  $a_{row,col} = a_{0,0} + (row * aligned\_row\_size + col) * item\_size$

Aligned memory management

- The CUDA class library have several functions to manage aligned memory. The following function allocates aligned memory area:

  *cudaMallocPitch(void** devPtr, size_t *pitch, size_t width, size_t height)*

  - *devPtr* – pointer to the aligned memory
  - *pitch* – alignment
  - *width* – size of one matrix row
  - *height* – number of matrix rows
- Similar to the linear memory management the start address of the allocated object will be stored in the devPtr variable
- The alignment is not an input value, this is one of the outputs of the function. The CUDA library will determine the optimal value (based on the array and device properties)
- Size of the matrix row is given by bytes

Copy aligned memory

- Because the different alignment the normal linear memory transfer is not usable in case of pitched memory regions
- The following CUDA function transfers data from one region to an other

  cudaMemcpy2D(void dst, size_t dpitch, const void* src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKing kind)*
  - *dst* – destination pointer
  - *dpitch* – destination pitch value
  - src – source pointer
  - *spitch* – source pith value
  - *width* – size of on row of the 2 dimensional array
  - *height* – number of rows of the 2 dimensional array
  - *kind* – transfer direction
    - host → host (*cudaMemcpyHostToHost*)
    - host → device (*cudaMemcpyHostToDevice*)
    - device → host (*cudaMemcpyDeviceToHost*)
    - device → host (*cudaMemcpyDeviceToDevice*)
- In case of simple not aligned arrays, the pitch value is 0

Matrix multiplication with aligned arrays

Exam

- Create CUDA application to solve the following problems: multiplying 2 dimensional (NxN) matrices with the GPU
  - N is a constant in source code
  - Allocate memory for 3 NxN matrices (A, B, C)
  - Fill the A matrix with numbers (for example: $a_{i,j} = i + j$)
  - Fill the B matrix with numbers (for example: $b_{i,j} = i - j$)
  - Allocate 3 NxN pitched arrays in the global memory of the graphics card (devA, devB, devC)
  - Move the input date to the GPU: A → devA, B → devB
  - Execute a kernel to calculate devC = devA * devB
  - Move the results back to the system memory: devC → C
  - List the values in the C vector to the screen

Kernel with aligned arrays

- The multiplier is the pitch value instead of the matrix column number
- The pitch size is given by bytes therefore in case of typed pointers we have to correct it's actual value by a sizeof(item_type) division
- devC = devA * devB source code:

```
1  __global__ static void MatrixMul(float *devA, float *devB, float *devC, size_t pitch) {
2     int indx = blockIdx.x * blockDim.x + threadIdx.x;
3     int indy = blockIdx.y * blockDim.y + threadIdx.y;
4
5     if (indx < N && indy < N) {
6        float sum = 0;
7        for(int i = 0; i < N; i++) {
8           sum += devA[indy * pitch/sizeof(float) + i] * devB[i * pitch/sizeof(float) + indx];
9        }
10       devC[indy * pitch/sizeof(float) + indx] = sum;
11    }
12 }
```

Invoke kernel with aligned arrays

- Initialization, allocate arrays

```
1  cudaSetDevice(0);
2  float A[N][N], B[N][N], C[N][N]; float *devA, *devB, *devC; size_t pitch;
3  cudaMallocPitch((void**) &devA, &pitch, sizeof(float) * N, N);
4  cudaMallocPitch((void**) &devB, &pitch, sizeof(float) * N, N);
5  cudaMallocPitch((void**) &devC, &pitch, sizeof(float) * N, N);
```

- Transfer input data (we assume pitch value is the same)

```
6  cudaMemcpy2D(devA, pitch, A, sizeof(float) * N, sizeof(float) * N, N, cudaMemcpyHostToDevice);
7  cudaMemcpy2D(devB, pitch, B, sizeof(float) * N, sizeof(float) * N, N, cudaMemcpyHostToDevice);
```

- Kernel invocation

```
8  dim3 grid((N - 1) / BlockN + 1, (N - 1) / BlockN + 1);
9  dim3 block(BlockN, BlockN);
10 MatrixMul<<<grid, block>>>(devA, devB, devC, pitch);
11 cudaThreadSynchronize();
```

- Transfer results, free memory

```
12 cudaMemcpy2D(C, sizeof(float) * N, devC, pitch, sizeof(float) * N, N, cudaMemcpyDeviceToHost);
13 cudaFree(devA); cudaFree(devB); cudaFree(devC);
```